# Proyecto 02 – Arquitectura DSA para Downscaling Bilineal
# Avance 4: Integración del Modo SIMD

Gabriel Guzmán Rojas  Sebastián Hernández Bonilla  José Vargas Torres

II Semestre 2025

## 1. Introducción

En el avance anterior se completó la implementación del modo secuencial de la unidad de interpolación bilineal (1 píxel por ciclo), incluyendo el manejo básico de memoria y la verificación funcional del módulo `ModoSecuencial`. De acuerdo con la planificación del proyecto, el objetivo de este avance es integrar el **modo SIMD** para procesar varios píxeles en paralelo ($N$ píxeles por ciclo), usando registros vectoriales (*SIMD registers*) y una máquina de estados finitos (FSM) que orquesta el flujo de datos.

En este documento se describe el diseño e integración de los siguientes bloques nuevos:

- `SIMD_Registros`: banco de registros vectoriales de entrada.
- `ModoSIMD`: arreglo paralelo de $N$ instancias del núcleo secuencial.
- `FSM_SIMD`: unidad de control que coordina carga, cómputo y finalización.
- `Top_SIMD`: módulo de integración que expone la interfaz de top–level.

Estos módulos preparan la arquitectura para realizar validación paralela y para comparar el rendimiento entre el modo secuencial y el modo SIMD en las siguientes etapas del proyecto.

## 2. Arquitectura general del modo SIMD

El enfoque seguido para el modo SIMD consiste en reutilizar el núcleo de interpolación `ModoSecuencial` como *building block* y replicarlo $N$ veces para procesar varios píxeles en paralelo. Cada núcleo sigue realizando el cálculo bilineal estándar a partir de los cuatro vecinos ($I00, I10, I01, I11$) y los coeficientes $\alpha$ y $\beta$ (valores derivados de la posición fraccionaria del píxel destino), pero ahora los datos se manejan como vectores:

- `I00_vec`$[0..N-1]$, `I10_vec`$[0..N-1]$, `I01_vec`$[0..N-1]$, `I11_vec`$[0..N-1]$.
- `alpha_vec`$[0..N-1]$, `beta_vec`$[0..N-1]$.
- `pixel_out_vec`$[0..N-1]$.

De esta forma, el modo SIMD mantiene exactamente la misma aritmética que el modo secuencial (misma precisión y mismo algoritmo), pero aumenta el *throughput* teórico en un factor aproximado de $N$, ignorando los ciclos de control y carga de datos.

En alto nivel, el flujo de datos del modo SIMD se puede describir así:

$$\text{Memoria} \rightarrow \texttt{SIMD\_Registros} \rightarrow \texttt{ModoSIMD} \rightarrow \text{Memoria/Salida},$$

coordinado por la FSM `FSM_SIMD` y encapsulado en `Top_SIMD`.

# 3. Diseño detallado de los módulos

## 3.1. `SIMD_Registros.sv`: banco de registros vectoriales

### Interfaz

El módulo `SIMD_Registros` es un banco de registros parametrizable que almacena un *batch* de $N$ píxeles de entrada y sus correspondientes coeficientes:

- Parámetro: `N` (cantidad de píxeles por batch).
- Entradas de control:
  - `clk`: reloj del sistema.
  - `rst`: *reset* síncrono que limpia todos los registros.
  - `load`: cuando vale 1 se captura el nuevo batch.
- Entradas de datos (vectores de tamaño $N$, cada elemento de 8 bits):
  - `I00_in[N]`, `I10_in[N]`, `I01_in[N]`, `I11_in[N]`.
  - `alpha_in[N]`, `beta_in[N]`.
- Salidas:
  - `I00_out[N]`, `I10_out[N]`, `I01_out[N]`, `I11_out[N]`.
  - `alpha_out[N]`, `beta_out[N]`.

Cada señal de salida tiene exactamente el mismo ancho y organización que la entrada correspondiente, pero ahora pasa por un registro interno.

### Comportamiento interno

Internamente, el módulo define arreglos de registros:

```
logic [7:0] I00_r[N]; I10_r[N]; I01_r[N]; I11_r[N]; alpha_r[N]; beta_r[N];
```

El comportamiento temporal es:

- En `rst = 1`, todos los arreglos se limpian a cero. Esto garantiza que, después de un reset, no existan valores basura en las entradas del bloque SIMD.
- Cuando `load = 1` en un flanco positivo de `clk`, se copian los vectores de entrada a los arreglos registrados.
- En el resto de los ciclos, si `load = 0`, los registros mantienen su valor anterior (comportamiento típico de registro síncrono tipo D).

Las salidas se conectan mediante `assign` directo a los arreglos internos (`I00_out = I00_r`, etc.), por lo que siempre reflejan el último batch cargado.

### Decisiones de diseño y ventajas

Este módulo concentra la función de **"congelar"** un conjunto de vectores de entrada para el bloque SIMD, desacoplando:

1. El momento en que la memoria o el controlador ponen los datos en el bus.
2. El momento en que el arreglo SIMD efectivamente los consume.

Esto simplifica la FSM de control (solo necesita generar un pulso de carga por batch) y evita que glitches o cambios en las señales externas durante el cómputo afecten el resultado de los núcleos en paralelo.

Además, al estar parametrizado en $N$, el banco de registros crece de forma transparente cuando se incrementa el ancho SIMD.

## 3.2. `ModoSIMD.sv`: arreglo SIMD de núcleos secuenciales

### Interfaz

El módulo `ModoSIMD` encapsula la parte **combinacional/secuencial** que realiza el cálculo bilineal en paralelo. Su interfaz es:

- Parámetro: `N`, cantidad de píxeles calculados por ciclo.
- Entradas:
  - `clk`, `rst`.
  - `valid_in`: pulso de arranque del batch, generado por la FSM.
  - Vectores de datos: `I00_vec[N]`, `I10_vec[N]`, `I01_vec[N]`, `I11_vec[N]`, `alpha_vec[N]`, `beta_vec[N]`.
- Salidas:
  - `valid_out`: indica que `pixel_out_vec` es válido.
  - `pixel_out_vec[N]`: píxeles interpolados en paralelo.

### Estructura interna: `generate` y reutilización de `ModoSecuencial`

La clave del diseño es el uso de un bucle `generate` para crear un **arreglo de núcleos**:

- Para cada índice `i` en $[0, N-1]$ se instancia `ModoSecuencial core_i`.
- A cada núcleo se le conecta:
  - Un escalar correspondiente del vector de entrada (por ejemplo, `I00_vec[i]`).
  - El mismo `valid_in` compartido entre todas las instancias.
- Cada núcleo produce su propio `pixel_out` y su propio `valid_out` interno, almacenados en arreglos como `pixel_int[i]` y `valid_int[i]`.

El valor de `pixel_out_vec[i]` se conecta a `pixel_int[i]`, de manera que cada posición del vector de salida corresponde directamente a una instancia del núcleo secuencial.

La señal global `valid_out` se toma de uno de los núcleos (la primera posición), asumiendo que todos tienen la misma latencia, lo cual es cierto porque todas las instancias comparten la misma lógica y la misma señal de arranque.

### Sincronización y latencia

Desde el punto de vista temporal:

- En el ciclo en que `valid_in` pasa de 0 a 1, cada núcleo `ModoSecuencial` captura sus entradas (que vienen de `SIMD_Registros`) y comienza el cálculo.
- Después de una cantidad fija de ciclos $L$ (latencia del núcleo secuencial), cada núcleo levanta su `valid_out` y coloca el píxel resultante en su salida.
- En ese mismo momento, `ModoSIMD` propaga el valor de `valid_int[0]` hacia `valid_out`, indicando que `pixel_out_vec` contiene un batch completo de $N$ píxeles.

Esta organización permite que todos los núcleos trabajen bloqueados (*lockstep*), lo que simplifica tanto la lógica como el análisis de *timing*.

### Decisiones de diseño

Los principales objetivos con este módulo fueron:

1. **Reutilizar** al máximo el código ya probado del modo secuencial, reduciendo el riesgo de introducir bugs.
2. Garantizar que el resultado SIMD sea *bit a bit* compatible con el resultado que se obtendría ejecutando el núcleo secuencial $N$ veces sobre la misma secuencia de datos.
3. Facilitar la escalabilidad: si en otra versión del proyecto se requiere, por ejemplo, pasar de $N = 4$ a $N = 8$, el cambio se limita al parámetro del módulo y a la disponibilidad de recursos en la FPGA.

## 3.3. `FSM_SIMD.sv`: máquina de estados de control

### Interfaz y propósito

El módulo `FSM_SIMD` implementa la lógica de control del flujo por batch. Su objetivo es que el modo SIMD se pueda ver desde afuera como un *bloque atómico*: se le entregan datos, se le da un `start` y, cuando la operación termina, se activa `done`.

Su interfaz es:

- Entradas:
  - `clk`, `rst`.
  - `start`: pulso para iniciar el procesamiento de un batch.
  - `simd_valid`: viene de `ModoSIMD.valid_out`.
- Salidas:
  - `load_regs`: habilita `SIMD_Registros` para capturar un nuevo batch de datos.
  - `run_simd`: pulso que se conecta a `valid_in` de `ModoSIMD`.
  - `write_back`: señal prevista para coordinar la escritura de resultados (en este avance queda como gancho para integración futura).
  - `done`: pulso de un ciclo que indica fin del batch.

### Secuencia de estados

Conceptualmente, la FSM recorre cinco fases:

1. **IDLE**: estado inicial. Todas las señales de control están en cero y se espera un pulso en `start`.
2. **LOAD**: se activa `load_regs` durante un ciclo para que los vectores de entrada que vienen del exterior se almacenen en `SIMD_Registros`.
3. **RUN**: se activa `run_simd` durante un ciclo, indicando a `ModoSIMD` que inicie el procesamiento del batch recién cargado.
4. **WAIT**: la FSM permanece en este estado hasta que `simd_valid` se pone en uno, lo que significa que `ModoSIMD` terminó el cálculo de los $N$ píxeles.
5. **WRITE**: durante un ciclo se activan `write_back` y `done`. Aquí se deja preparado el sistema para eventualmente escribir los resultados en memoria o entregarlos a otro bloque. Después de este estado se vuelve a **IDLE**.

Esta secuencia implementa un *handshake* claro entre el entorno externo, los registros y el bloque SIMD.

### Beneficios del enfoque por FSM

Al encapsular la lógica de control en una FSM dedicada se logra:

- Mantener `ModoSIMD` puramente concentrado en el cálculo, sin lógica de protocolo alrededor.
- Facilitar la integración con controladores más grandes (por ejemplo, el controlador global del DSA o un bus AXI-lite) que solo necesitan interactuar con señales abstractas `start` y `done`.
- Permitir, en futuros avances, extender fácilmente la FSM para soportar múltiples batches encadenados, contadores de ciclos, manejo de errores, etc.

## 3.4. `Top_SIMD.sv`: integración y vista de sistema

### Interfaz de alto nivel

El módulo `Top_SIMD` es la capa de integración que junta todos los bloques anteriores y define la interfaz que se conectará al resto del sistema.

- Parámetro: `N` (ancho SIMD).
- Entradas:
  - `clk`, `rst`.
  - `start`: pulso que inicia el procesamiento de un batch.
  - Vectores de entrada de tamaño $N$: `I00_vec`, `I10_vec`, `I01_vec`, `I11_vec`, `alpha_vec`, `beta_vec`.
- Salidas:
  - `done`: indica que el batch actual ha terminado.
  - `pixel_out_vec[N]`: píxeles procesados en paralelo.

### Señales internas y conexiones

`Top_SIMD` define señales internas que unen a los distintos submódulos:

- Señales de control:
  - `load_regs`, `run_simd`, `write_back`, `simd_valid`.
- Vectores registrados:
  - `I00_reg[N]`, `I10_reg[N]`, `I01_reg[N]`, `I11_reg[N]`, `alpha_reg[N]`, `beta_reg[N]`.

La integración se realiza en tres pasos:

1. **FSM_SIMD**: recibe `start` y `simd_valid`, y produce `load_regs`, `run_simd`, `done` y `write_back`.
2. **SIMD_Registros**: recibe las entradas externas y la señal `load_regs`. Cuando esta se activa, captura el batch en los vectores registrados `Ixx_reg` y `alpha_reg`/`beta_reg`.
3. **ModoSIMD**: usa los vectores registrados como entrada, se activa con `run_simd` y devuelve `pixel_out_vec` y `simd_valid`. Esta última cierra el ciclo de control con la FSM.

### Rol dentro del proyecto

Con `Top_SIMD` se consigue un bloque compacto que, desde el punto de vista del sistema, se comporta como un *acelerador SIMD* parametrizable:

- Recibe un conjunto de $N$ píxeles fuente y sus coeficientes.
- Procesa todo en paralelo disparando un núcleo por elemento.
- Entrega $N$ píxeles destino y avisa cuándo el resultado está listo.

Esta abstracción facilita las etapas siguientes del proyecto:

- Conexión con los módulos que leen bloques de la imagen de entrada y escriben la imagen de salida.
- Instrumentación de contadores de ciclos para comparar *performance* secuencial vs. SIMD.
- Posible replicación de este top para construir arquitecturas más complejas (por ejemplo, varios bloques SIMD trabajando en paralelo sobre distintas ventanas de la imagen).

# 4. Módulos de downscale secuencial y SIMD

## 4.1. Downscale_Secuencial.sv

El módulo `Downscale_Secuencial` implementa el proceso completo de *downscaling* bilineal de una imagen de entrada SRC_H × SRC_W hacia una imagen de salida DST_H × DST_W procesando **un píxel por vez**. El diseño es parametrizable en las dimensiones de la imagen:

- Parámetros: `SRC_H`, `SRC_W`, `DST_H`, `DST_W`.
- Entradas:
  - `clk`, `rst`, `start`.
  - `image_in[0:SRC_H-1][0:SRC_W-1]`: imagen de entrada de 8 bits por píxel.
- Salidas:
  - `done`: indica que toda la imagen de salida fue generada.
  - `image_out[0:DST_H-1][0:DST_W-1]`: imagen reducida.

Internamente se calculan los ratios en formato Q8.8:

$$\texttt{X\_RATIO\_FP} = \frac{(\texttt{SRC\_W} - 1) \ll 8}{\texttt{DST\_W} - 1}, \quad \texttt{Y\_RATIO\_FP} = \frac{(\texttt{SRC\_H} - 1) \ll 8}{\texttt{DST\_H} - 1}$$

que permiten obtener la posición fuente $(x_{\text{src}}, y_{\text{src}})$ para cada píxel destino usando aritmética fija.

El módulo instancia el interpolador `ModoSecuencial`, al que le entrega:

- Los cuatro vecinos `I00`, `I10`, `I01`, `I11` tomados de `image_in` según las coordenadas calculadas.
- Los pesos fraccionales `alpha` y `beta` (partes fraccionarias de $x_{\text{src}}$ y $y_{\text{src}}$ en Q0.8).
- Un pulso `valid_in` cada vez que se desea calcular un nuevo píxel.

La lógica de control se implementa con una FSM simple:

- **S_IDLE**: espera `start`. Cuando se activa, limpia la imagen de salida y posiciona los índices destino `i_dst`, `j_dst` en $(0, 0)$.
- **S_SETUP**: para el píxel destino actual calcula $x_{\text{src}}$, $y_{\text{src}}$, determina `x_l`, `x_h`, `y_l`, `y_h`, lee los vecinos de `image_in` y arma las señales `I00..I11`, `alpha`, `beta`. Luego genera `valid_in` y pasa a **S_WAIT_RESULT**.
- **S_WAIT_RESULT**: espera a que `valid_out` del `ModoSecuencial` se active. Cuando ocurre, almacena `pixel_out` en `image_out[i_dst][j_dst]` y actualiza los índices destino:
  - Si todavía no es el último píxel $(\texttt{DST\_H} - 1, \texttt{DST\_W} - 1)$, incrementa `j_dst` (y `i_dst` cuando termina una fila) y regresa a **S_SETUP**.
  - Si ya se procesó el último píxel, activa `done` y pasa a **S_DONE**.
- **S_DONE**: mantiene la señal `done` en alto hasta que `start` vuelva a cero, momento en que regresa a **S_IDLE**.

Este módulo realiza, por lo tanto, todo el barrido de la imagen destino en forma secuencial, aplicando la interpolación bilineal píxel a píxel encima del núcleo `ModoSecuencial`.

## 4.2. Downscale_SIMD.sv

El módulo Downscale_SIMD extiende la idea anterior al caso **paralelo**, usando el bloque Top_SIMD que procesa $N$ píxeles en cada batch. Sus parámetros son análogos a los del módulo secuencial, pero añade N como número de *lanes*:

- Parámetros: SRC_H, SRC_W, DST_H, DST_W, N.
- Entradas: clk, rst, start, image_in.
- Salidas: done, image_out.

Al igual que en el caso secuencial, se calculan X_RATIO_FP y Y_RATIO_FP en Q8.8, pero ahora el sistema recorre la imagen de salida en bloques de $N$ píxeles consecutivos. Para ello se define:

- TOT_PIX = DST_H * DST_W: cantidad total de píxeles destino.
- base_idx: índice lineal del primer píxel del batch actual.
- Arreglos por lane: idx[g], i_dst[g], j_dst[g], x_src_fp[g], y_src_fp[g], x_l[g], x_h[g], y_l[g], y_h[g] y valid_lane[g].

Mediante un bloque generate se define la lógica de cada lane:

- **Cálculo del índice y coordenadas**: idx[g] = base_idx + $g$. Si idx[g] < TOT_PIX, el lane está activo y se calcula:

$$i_{\text{dst}}[g] = \texttt{idx[g]}/\texttt{DST\_W}, \quad j_{\text{dst}}[g] = \texttt{idx[g]} \bmod \texttt{DST\_W}$$

- **Posición fuente y vecinos**: A partir de $i_{\text{dst}}$ y $j_{\text{dst}}$ se calculan $x_{\text{src}}$ y $y_{\text{src}}$ en Q8.8, se obtienen x_l, y_l, y las versiones x_h, y_h con saturación en los bordes. Con estos índices se leen I00, I10, I01, I11 desde image_in.
- **Pesos fraccionales**: Las partes fraccionarias de x_src_fp y y_src_fp se convierten a alpha_vec[g] y beta_vec[g] en Q0.8.
- Si el lane es inactivo (idx[g] fuera de rango), se asignan ceros para evitar accesos inválidos.

Para el cálculo de los píxeles, Downscale_SIMD instancia Top_SIMD y le entrega, en cada batch, los vectores I00_vec, I10_vec, I01_vec, I11_vec, alpha_vec y beta_vec de tamaño $N$, junto con un pulso de inicio top_start. Cuando Top_SIMD finaliza el procesamiento del batch (señal top_done), el módulo Downscale_SIMD escribe los resultados de pixel_out_vec[g] en sus respectivas posiciones image_out[i_dst[g]][j_dst[g]].

El flujo de control está gobernado por una FSM con los estados:

- **S_IDLE**: espera start, limpia contadores y reinicia base_idx.
- **S_PREP_BATCH**: prepara los datos del batch llenando los vectores Ixx_vec y alpha/beta_vec para cada lane válido.
- **S_START_TOP**: genera un pulso de inicio para Top_SIMD.
- **S_WAIT_TOP**: espera top_done (fin del cálculo paralelo de los $N$ píxeles).
- **S_WRITE_BATCH**: escribe los resultados del batch en la imagen de salida y actualiza base_idx sumando $N$. Si aún quedan píxeles por procesar, vuelve a **S_PREP_BATCH**; si ya se procesaron los TOT_PIX, pasa a **S_DONE**.
- **S_DONE**: mantiene done alto hasta que start vuelva a cero.

Con este esquema, el módulo procesa la imagen de salida en bloques de $N$ píxeles, explotando el paralelismo del arreglo SIMD y reduciendo significativamente el número total de ciclos respecto al diseño secuencial.

# 5. Validación paralela y contraste entre los testbench Secuencial y SIMD

El objetivo principal de esta sección es contrastar directamente las ejecuciones de los dos bancos de pruebas en modo *proceso* —el secuencial y el SIMD— para demostrar dos puntos fundamentales:

1. Que ambos módulos producen los mismos resultados píxel a píxel (validación paralela funcional).
2. Que el módulo `Downscale_SIMD` reduce significativamente los ciclos de ejecución en comparación con `Downscale_Secuencial`, validando el paralelismo real de la arquitectura SIMD.

Ambos testbench procesan la misma imagen fuente de $32 \times 32$ píxeles, generando una imagen destino de $16 \times 16$ píxeles (256 píxeles totales). Además, ambos construyen una matriz de referencia `expected[i][j]` utilizando una función bilineal exacta en punto flotante, de manera que todas las comparaciones entre hardware y referencia sean consistentes.

## 5.1. Ejecución del testbench secuencial

El archivo `tb_downscale_Secuencial_Proceso.sv` valida el módulo `Downscale_Secuencial`, el cual procesa **un píxel destino por vez**. Su comportamiento se resume así:

- Para cada uno de los 256 píxeles destino, el módulo realiza:
  - Cálculo de coordenadas fuente.
  - Obtención de vecinos.
  - Cálculo de pesos $\alpha$ y $\beta$.
  - Ejecución del interpolador `ModoSecuencial`.
- El testbench espera la señal `valid_out` por cada píxel.
- Cada resultado de hardware se compara con `expected[i][j]`.

El testbench imprime un resumen final indicando:

```
PASS = 256, FAIL = 0
```

lo cual demuestra que el módulo secuencial genera correctamente todos los píxeles con una tolerancia de $\pm 1$ LSB.

Además, el testbench registra el número total de ciclos:

$$C_{\text{seq}} = \textit{(obtenido de la simulación)}$$

## 5.2. Ejecución del testbench SIMD

El archivo `tb_downscale_SIMD_Proceso.sv` valida el módulo `Downscale_SIMD`, configurado para procesar $\mathbf{N = 4}$ **píxeles en paralelo** en cada batch.

El comportamiento observado es:

- Para cada batch, se preparan cuatro píxeles destino consecutivos.
- Se obtiene para cada lane:
  - Coordenadas fuente,
  - Vecinos,
  - Pesos $\alpha$ y $\beta$.
- Los cuatro píxeles se procesan simultáneamente mediante `Top_SIMD`.
- Cada salida del vector `pixel_out_vec` se coloca en su posición correspondiente en `image_out`.

Al finalizar, el testbench reporta el rendimiento del módulo:

$$C_{\text{simd}} = 513 \text{ ciclos}, \qquad T_{\text{simd}} = 5130 \text{ ns}$$

Y, al igual que en el secuencial:

`PASS = 256, FAIL = 0`

lo cual confirma que el módulo SIMD replica exactamente el comportamiento del módulo secuencial, pero en menor tiempo.

### 5.3. Validación paralela: igualdad píxel a píxel

La validación paralela consiste en verificar que:

$$\texttt{image\_out\_sec[i][j]} = \texttt{image\_out\_simd[i][j]} \quad \forall i, j$$

Ambos testbench demostraron:

- Los 256 valores generados por el módulo secuencial coinciden con la referencia.
- Los 256 valores generados por el módulo SIMD coinciden también con la referencia.

Por lo tanto, las dos arquitecturas entregan la misma imagen reducida y los resultados del módulo SIMD son numéricamente correctos.

## 6. Resultados y comparación de rendimiento

Esta sección presenta los resultados obtenidos en la simulación completa de los módulos `Downscale_Secuencial` y `Downscale_SIMD` utilizando sus respectivos testbench en modo *proceso*. Ambos módulos procesaron la misma imagen fuente de $32 \times 32$ píxeles y generaron una imagen destino de $16 \times 16$ píxeles, equivalente a un total de 256 resultados a validar provenientes de dos testbench diferentes enfocados a rendimiento.

### 6.1. Resultados del módulo secuencial

El testbench `tb_Downscale_Secuencial.sv` ejecutó el recorrido completo de la imagen, realizando un cálculo bilineal por cada píxel destino. Los valores fueron comparados contra la matriz de referencia generada por el modelo en punto flotante.

Los resultados obtenidos fueron:

- **Ciclos totales:** 769
- **Tiempo total:** 7690 ns (periodo de 10 ns)
- **Exactitud:** `PASS = 256`, `FAIL = 0`

El 100 % de los píxeles coincidieron con la referencia dentro de la tolerancia de $\pm 1$ LSB, confirmando la correcta operación del diseño secuencial.

### 6.2. Resultados del módulo SIMD

El testbench `tb_Downscale_SIMD.sv` probó el módulo en configuración $\mathbf{N = 4}$ **lanes**, procesando cuatro píxeles destino en paralelo durante cada batch de cómputo.

Los resultados fueron:

- **Ciclos totales:** 513
- **Tiempo total:** 5130 ns (periodo de 10 ns)
- **Exactitud:** `PASS = 256`, `FAIL = 0`

Al igual que en el diseño secuencial, todos los píxeles coincidieron con la referencia, demostrando que el paralelismo no modifica la exactitud numérica.

## 6.3. Comparación directa del rendimiento

Ambos módulos producen exactamente la misma imagen de salida, pero con costos temporales muy diferentes. La siguiente tabla resume el contraste entre ambas implementaciones:

| Implementación | Ciclos totales | Tiempo [ns] | Aceleración |
|---|---|---|---|
| Secuencial | 769 | 7690 | 1,0× |
| SIMD ($N = 4$ lanes) | 513 | 5130 | $\frac{769}{513} \approx 1{,}50\times$ |

Cuadro 1: Comparación de rendimiento entre las versiones Secuencial y SIMD.

A partir de estos datos se observa:

- El módulo SIMD reduce el número total de ciclos en **un 33.3 %**.
- El tiempo total disminuye de 7690 ns a 5130 ns.
- La aceleración medida es:
$$\text{speedup} = \frac{769}{513} \approx 1{,}50\times$$
- Aunque $N = 4$ lanes SIMD permitiría un speedup teórico de 4×, el speedup real es menor debido a:
  - ciclos de preparación del batch,
  - ciclos de avance de la FSM,
  - redondeo en los límites de la imagen destino,
  - ciclos finales de escritura y señalización del `done`.

## 6.4. Conclusión de los resultados

La experimentación demuestra que:

1. Ambos módulos generan exactamente los mismos 256 valores de salida.
2. La implementación SIMD es funcionalmente equivalente al diseño secuencial.
3. El paralelismo SIMD logra una aceleración real de 1,50× para $N = 4$ lanes.

Estos resultados validan que el paralelismo introducido en el módulo SIMD es correcto y ofrece una mejora tangible de rendimiento sin comprometer la exactitud.